HW5 Classifition of digits

January 7, 2022

[1]:
```
import astropy.io.fits as pf
import numpy as np
import matplotlib.pyplot as plt
import torch
from skimage.transform import rotate
```

[2]:
```
def give_me_Pytorch_data_form(input_image, input_labels,torch_format=True):
    whole_length = len(input_image[0])
    width = int(np.sqrt(whole_length))
    image_matrix = np.zeros((len(input_image),1,width,width))
    labels_matrix = np.zeros((len(input_image),10))
    for i in range(0,len(input_image)):
        image_matrix[i,0,:,:]=input_image[i].reshape(width,width)
        labels_matrix[i,input_labels[i]]=1.

    if torch_format==True:
        image_matrix = torch.from_numpy(np.array(image_matrix,dtype=np.float32))
        labels_matrix = torch.from_numpy(np.array(labels_matrix,dtype=np.
↪float32))

    return image_matrix,labels_matrix
```

[3]:
```
train = pf.open('Training.fits')
test = pf.open('Test.fits')
```

[4]:
```
image_train, image_labels = give_me_Pytorch_data_form(train[0].data,train[1].
↪data,torch_format=True)
image_test, test_labels = give_me_Pytorch_data_form(test[0].data,test[1].data)
```

[36]:
```
class CNN(torch.nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        ###
        ### input 1x8x8
        self.cnn1 = torch.nn.Sequential(
            torch.nn.Conv2d(in_channels=1,␣
↪out_channels=8,kernel_size=3,padding=1,stride=1), #8x8x8
```

```
            torch.nn.BatchNorm2d(8),
            #torch.nn.MaxPool2d(kernel_size=2), #20x4x4
            torch.nn.ReLU(),
            )
        self.cnn2 = torch.nn.Sequential(
            torch.nn.Conv2d(in_channels=8,␣
↪out_channels=8,kernel_size=3,padding=1,stride=1),#8x8x8
            torch.nn.BatchNorm2d(8),
            #torch.nn.MaxPool2d(kernel_size=2), #20x4x4
            torch.nn.ReLU(),

        )
        self.cnn3 = torch.nn.Sequential(
            torch.nn.Conv2d(in_channels=8,␣
↪out_channels=8,kernel_size=3,padding=1,stride=1),#8x8x8
            torch.nn.BatchNorm2d(8),
            #torch.nn.MaxPool2d(kernel_size=2), #20x4x4
            torch.nn.ReLU(),

        )

        self.fc1 = torch.nn.Sequential(
            torch.nn.Linear(512, 300)
           )
        self.fc2 = torch.nn.Sequential(
             torch.nn.Linear(300, 10)
           )

        #self.fc2 = torch.nn.Linear(10, 10)
    def forward(self, x):
        x = self.cnn1(x)
        x = self.cnn2(x)
        x = self.cnn3(x)
        x = x.view(x.size(0), -1)
        x = self.fc1(x)
        x = self.fc2(x)
        out = x
        return out
```

[37]:
```
model = CNN()
```

[38]:
```
learningRate = 0.01
epochs = 100
criterion = torch.nn.CrossEntropyLoss()
# Just the loss function : here we use the default CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=learningRate)
```

[39]:
```
accuracy_array = []
accuracy_test_array = []
loss_array = []
loss_test_array = []
epoch_array = []
N_total_train = len(image_train)
batch_size=64
for epoch in range(epochs):

    model.train() # here is to tell the code to do the training again after␣
↪model.eval() mode
    # Clear gradient buffers because we don't want any gradient from previous␣
↪epoch to carry forward, dont want to cummulate gradients
    for start_index_batch in range(0,N_total_train,batch_size):
        # Clear gradient buffers because we don't want any gradient from␣
↪previous epoch to carry forward, dont want to cummulate gradients
        optimizer.zero_grad()
        end_index = min(start_index_batch + batch_size, N_total_train)
        # get output from the model, given the inputs
        #print(len(color_train[start_index_batch:end_index,:]))
        outputs = model(image_train[start_index_batch:end_index])

        # get loss for the predicted output
        loss = criterion(outputs, image_labels[start_index_batch:end_index,:])
        # get gradients w.r.t to parameters
        loss.backward()
        # update parameters
        optimizer.step()

    model.eval() # here is to fix the parameters (mean, sigma) of the␣
↪batchnormalization, given that the batchnorm calculates the mean sigma with a␣
↪new batch everytime during training.
    with torch.no_grad():
        outputs_all = model(image_train)
        pred_y = torch.max(outputs_all, 1)[1].data.squeeze()
        accuracy = torch.sum((pred_y == torch.max(image_labels, 1)[1].data.
↪squeeze()) / pred_y.size(0))
        epoch_array.append(epoch)

        loss_array.append(float(loss.detach().numpy()))
        accuracy_array.append(float(accuracy.numpy()))


        outputs_test = model(image_test)
        loss_test = criterion(outputs_test, test_labels)

        pred_y_test = torch.max(outputs_test, 1)[1].data.squeeze()
```

```
        accuracy_test = torch.sum((pred_y_test == torch.max(test_labels, 1)[1].
↪data.squeeze()) / pred_y_test.size(0))
        accuracy_test_array.append(float(accuracy_test.numpy()))
        loss_test_array.append(float(loss_test.detach().numpy()))

    if epoch % 1 ==0:
        print(epoch,accuracy.numpy(),loss.detach().numpy(), accuracy_test.
↪numpy(),loss_test.detach().numpy())
```

```
0 0.6674091 1.9089136 0.6533333 1.9982228
1 0.81143296 1.26709 0.7999999 1.4118378
2 0.86414266 0.7762973 0.8755555 1.0080646
3 0.89309585 0.46784624 0.8955554 0.74801
4 0.91462517 0.30049518 0.91777766 0.58092326
5 0.9302154 0.20782883 0.92888874 0.47090495
6 0.94060886 0.15414941 0.9377776 0.3951376
7 0.9524871 0.12097462 0.9422221 0.34113452
8 0.9576839 0.09760174 0.95111096 0.29943997
9 0.9621383 0.0805796 0.95111096 0.26539725
10 0.96956223 0.068254076 0.95555544 0.23855467
11 0.971047 0.05863067 0.9577776 0.21681021
12 0.9732741 0.051132888 0.9622221 0.19829518
13 0.9747589 0.045703143 0.96888864 0.18259545
14 0.97698605 0.040732384 0.97111094 0.16925932
15 0.9777284 0.03667055 0.97111094 0.15784451
16 0.9784708 0.03324698 0.9755554 0.14798434
17 0.980698 0.030257314 0.9755554 0.13924143
18 0.98144037 0.027381934 0.9777776 0.13128805
19 0.9829252 0.025176233 0.9777776 0.12459312
20 0.9844099 0.023179403 0.97999984 0.11850242
21 0.98515236 0.021458661 0.97999984 0.11288742
22 0.9858948 0.019943198 0.97999984 0.107867084
23 0.9858948 0.018672096 0.982222 0.103345305
24 0.9858948 0.01750827 0.98444426 0.099330515
25 0.9866372 0.0164232 0.98444426 0.095639884
26 0.98737955 0.015459518 0.98444426 0.092411585
27 0.98737955 0.014556766 0.9866665 0.08922772
28 0.9881219 0.013696298 0.9866665 0.08637517
29 0.9896067 0.012946651 0.9866665 0.08378974
30 0.99034905 0.0122980615 0.9866665 0.08136852
31 0.99183387 0.011614793 0.9866665 0.0790262
32 0.99406105 0.0110470755 0.9866665 0.07697936
33 0.99406105 0.010472099 0.9866665 0.07501897
34 0.99406105 0.009996786 0.9866665 0.07313783
35 0.99406105 0.009543107 0.98888874 0.071489505
36 0.9948035 0.009089956 0.98888874 0.069890745
37 0.99554586 0.008690859 0.98888874 0.068352625
38 0.99628824 0.008292768 0.98888874 0.06697466
```

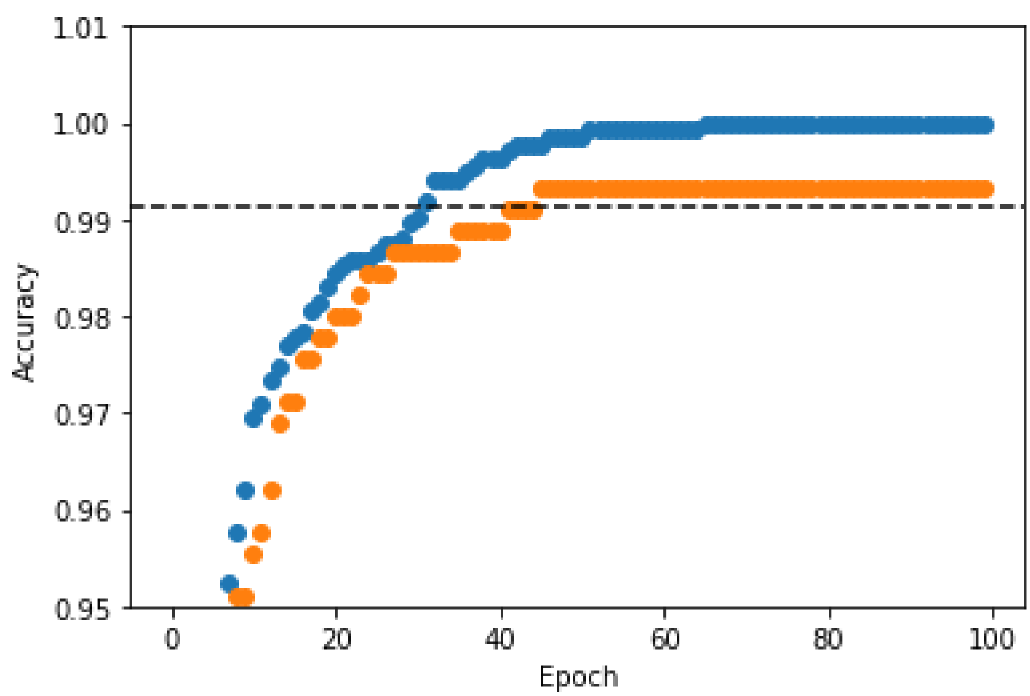
39 0.99628824 0.00798048 0.98888874 0.06567833
40 0.99628824 0.007676914 0.98888874 0.06446539
41 0.9970307 0.007340388 0.991111 0.06330159
42 0.99777305 0.007033352 0.991111 0.06220348
43 0.99777305 0.0067479922 0.991111 0.061215453
44 0.99777305 0.0065053147 0.991111 0.060236957
45 0.99777305 0.006258952 0.9933332 0.059425335
46 0.9985154 0.006048886 0.9933332 0.058575418
47 0.9985154 0.0058182254 0.9933332 0.057746273
48 0.9985154 0.0056066695 0.9933332 0.05693361
49 0.9985154 0.0054134578 0.9933332 0.05624282
50 0.9985154 0.005279811 0.9933332 0.055474468
51 0.99925786 0.0050948733 0.9933332 0.054809608
52 0.99925786 0.0049358676 0.9933332 0.054173753
53 0.99925786 0.0048235473 0.9933332 0.053588178
54 0.99925786 0.004664052 0.9933332 0.05301433
55 0.99925786 0.0044960906 0.9933332 0.052401636
56 0.99925786 0.00437836 0.9933332 0.0518725
57 0.99925786 0.004239598 0.9933332 0.051369183
58 0.99925786 0.0041139224 0.9933332 0.05085205
59 0.99925786 0.0040082275 0.9933332 0.050385036
60 0.99925786 0.0038967011 0.9933332 0.049964216
61 0.99925786 0.0037848165 0.9933332 0.04951865
62 0.99925786 0.0036700256 0.9933332 0.049117588
63 0.99925786 0.0035902031 0.9933332 0.048735067
64 0.99925786 0.0034765035 0.9933332 0.04836196
65 1.0000002 0.0033947127 0.9933332 0.04801236
66 1.0000002 0.0033038547 0.9933332 0.04767385
67 1.0000002 0.00323167 0.9933332 0.04733122
68 1.0000002 0.0031587696 0.9933332 0.04700749
69 1.0000002 0.003095702 0.9933332 0.04669368
70 1.0000002 0.0030304946 0.9933332 0.046403013
71 1.0000002 0.002940877 0.9933332 0.046097938
72 1.0000002 0.002892971 0.9933332 0.045833465
73 1.0000002 0.0028171632 0.9933332 0.045545336
74 1.0000002 0.0027699664 0.9933332 0.04527895
75 1.0000002 0.0027092434 0.9933332 0.045031343
76 1.0000002 0.0026443005 0.9933332 0.04475741
77 1.0000002 0.0025903673 0.9933332 0.04453938
78 1.0000002 0.0025397185 0.9933332 0.044279188
79 1.0000002 0.0024934157 0.9933332 0.044069976
80 1.0000002 0.0024354467 0.9933332 0.04384378
81 1.0000002 0.0023888622 0.9933332 0.043617304
82 1.0000002 0.002337612 0.9933332 0.043434888
83 1.0000002 0.0023043114 0.9933332 0.043234725
84 1.0000002 0.0022554991 0.9933332 0.043054026
85 1.0000002 0.0022066084 0.9933332 0.04284404
86 1.0000002 0.002159735 0.9933332 0.042688165

```
87 1.0000002 0.0021186334 0.9933332 0.042499315
88 1.0000002 0.0020746847 0.9933332 0.042344812
89 1.0000002 0.002037263 0.9933332 0.042153273
90 1.0000002 0.0019961966 0.9933332 0.042010903
91 1.0000002 0.0019590512 0.9933332 0.041846495
92 1.0000002 0.0019260948 0.9933332 0.04171504
93 1.0000002 0.0018954776 0.9933332 0.041547704
94 1.0000002 0.0018549188 0.9933332 0.04141969
95 1.0000002 0.001827268 0.9933332 0.04127104
96 1.0000002 0.001792881 0.9933332 0.041160394
97 1.0000002 0.0017630464 0.9933332 0.041037615
98 1.0000002 0.0017326974 0.9933332 0.040916935
99 1.0000002 0.0017050795 0.9933332 0.040805068
```

[40]:

```
plt.scatter(epoch_array,accuracy_array,color='C0')
plt.scatter(epoch_array,accuracy_test_array,color='C1')
plt.ylim(0.95,1.01)
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.axhline(0.9912,ls='--',color='black')
```

[40]: <matplotlib.lines.Line2D at 0x1454ae0a0>



[ ]: